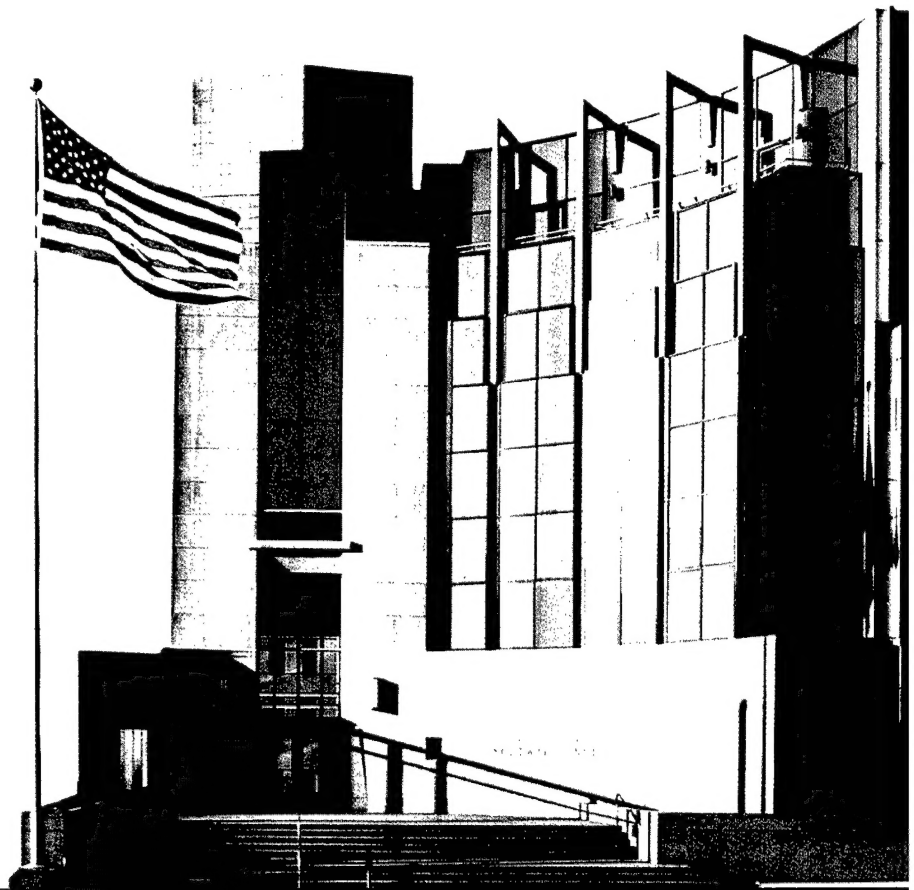
Carnegie Mellon
Software Engineering Institute

Snapshot of CCL: A Language for Predictable Assembly

Kurt C. Wallnau
James Ivers

June 2003

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited



TECHNICAL NOTE
CMU/SEI-2003-TN-025

Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

Snapshot of CCL: A Language for Predictable Assembly

CMU/SEI-2003-TN-025

Kurt C. Wallnau
James Ivers

June 2003

Predictable Assembly from Certifiable Components Initiative

Table of Contents

List of Figures

Abstract

Construction and composition language (CCL) plays several roles in our approach to achieving automated predictable assembly. CCL is used to produce specifications that contain structural, behavioral, and analysis-specific information about component technologies, as well as components and assemblies in such technologies. These specifications are translated to one or more reasoning frameworks that analyze and predict the runtime properties of assemblies. CCL processors can also be used to automate many of the constructive activities of component-based development through various forms of program generation. Using a common specification for prediction and construction improves confidence that analysis models match implementations. This report presents a snapshot of CCL by examining a small example CCL specification.

1 Introduction

The Predictable Assembly from Certifiable Components (PACC) Initiative at the Software Engineering Institute is investigating technology and methods for reliably predicting the run-time behavior of assemblies of components from their certifiable properties. Our approach to achieving predictable assembly is based on the development and use of prediction-enabled component technologies (PECTs).

A PECT extends the notion of a component technology with one or more reasoning frameworks such that assemblies of components (or simply "assemblies") are predictable with respect to those frameworks. When reasoning about components and assemblies, we start with their specifications rather than directly with their implementations. These specifications contain structural information (how components are arranged to form assemblies), behavioral information (how components interact with each other), and analysis-specific properties (e.g., execution times and thread priorities for a performance reasoning framework). The construction and composition language (CCL) is a language for writing such specifications.

CCL specifications play a central role in using a PECT to make predictions. They are the source of all component and assembly information that is needed by reasoning frameworks and are used to automate important tasks involved in making predictions, such as

- checking that an assembly conforms to the rules of the component technology
- checking that an assembly conforms to the rules of the reasoning frameworks that will be used to make predictions
- interpreting the CCL specification to the forms needed by different reasoning frameworks

CCL specifications also play a role in ensuring that predictions are valid. Since any prediction can be only as good as the information on which it is based and such information comes from CCL specifications, certifying that the information in a CCL specification is correct is an important step. Doing so can be accomplished in different ways, such as

- generating an implementation (or significant parts of an implementation) from a CCL specification
- extracting the CCL specification directly from the implementation
- testing an implementation for conformance to the information found in a corresponding CCL specification

This report is a snapshot of CCL, illustrating the main concepts of the language by working through a simple example. The concepts on which CCL is based (e.g., components, pins, reactions, and interactions) are defined in *Volume III: A Technology for Predictable Assembly from Certifiable Components* [Wallnau 03]. While Vol. III provides essential background material, the basic ideas of this report should be reasonably clear to anyone who is familiar with software component technology or with the most recent version of the Unified Modeling Language—UML2.0.[1]

1.1 About This Technical Note

Our primary objective for writing this technical note is to provide a snapshot of CCL and how it captures information needed to achieve predictable assembly so that we can elicit feedback on both the direction the language is taking and its form. As such, the primary audience of this report consists of those also working to achieve predictable assembly, who may be users of CCL at some point. Familiarity with the evolution of this work, including an earlier version of the language (CL), is not necessary, but may provide additional insight into the rationale for some features of CCL [Ivers 02, Wallnau 03, Ivers 03].

1.2 Notational Conventions

CCL text is presented in `11 point Courier`. CCL keywords are shown in **`boldface Courier`**. We *use italics* to refer to a CCL concept rather than to a specific phrase in CCL. Terms that are defined in the glossary are underlined. The graphical notation used in this report is introduced in Vol. III [Wallnau 03], but should not be mistaken for a rigorously defined notation. The semantics of diagrams in this notation are shown in the accompanying CCL description of each figure, except for the first, whose corresponding CCL is found in Appendix A. A particularly important convention is the graphical distinction between instances and types—boxes with solid lines denote instances, while boxes with dashed lines denote types.

1.3 Organization of This Technical Note

The remainder of this technical note is organized around the explanation of how CCL is used to model an example. This explanation is provided in Section 2, which begins with a rough explanation of an example that is incrementally elaborated and specified in CCL in subsequent subsections. Although the example is not intended to be a complete exposition of CCL, it

1. A reasonable question at this point is "if the concepts are similar, why not just use UML?" There are two reasons. First, we are still investigating our needs, and experimenting with language features to try out different ideas, which would not be at all practical if constrained by UML and its associated tools. Second, UML is currently evolving, albeit in a useful and relevant way. While UML 2.0 is not yet official, we see how we could map CCL concepts easily to UML 2.0's additions for modeling architectural information. When UML 2.0 is official and tool support is available, we anticipate a smooth migration path to UML.

introduces most of the important aspects of the language. For some features of CCL not covered by the example, we have provided self-contained capsule summaries. Section 4 concludes the report with a brief discussion of the current state of the language and our thoughts on the next steps in its development.

2 Example Application Assembly

The main features of CCL are exhibited in the assembly depicted in Figure 1. To summarize what is known from the graphic

- The assembly `myApp` executes in a runtime environment `appEnv`.
- `appEnv` provides services `appClock` and `appLog`.
- `myApp` comprises component `c3` and partial assembly `inner`.
- `inner` interacts through its `drip` pin with `c3` through its `toc` pin.
- `inner` and `myApp` use the same `appClock` and `appLog` services provided by `appEnv`.
- `c3` interacts with `appLog`.

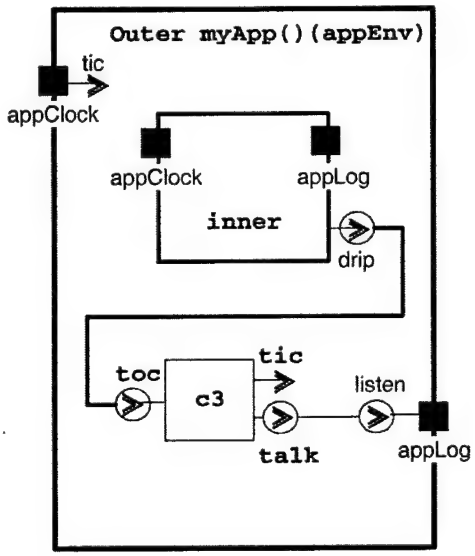


Figure 1: Example Assembly

A graphical notation is effective at conveying some kinds of design information, though often only when the quantity of information is small. The graphic in Figure 1 abstracts all sorts of information, such as

- `inner` uses `appClock` to initiate the behavior of `myApp`.
- Events and different kinds of data are exchanged on interactions (on "the wire").

- Interactions conform to well-defined protocols (e.g., to enforce event-queuing policies).
- Components react to stimuli and stimulate other components in well-defined ways.

To specify these and other aspects of a predictable assembly requires richer detail than is conveniently expressed with graphic notation alone; this is where the textual CCL fits. The example presented in this technical note is meant to give the reader an overall impression of CCL. While the language is still under development and some details are likely to change, the major concepts of CCL are likely to remain stable.

The example is incrementally explained in five successive steps, each of which is presented in some detail in the following sections. The complete CCL specification of the example is provided in Appendix A.

2.1 Specify the Component

In the vernacular of component-based development, the term "component" is used (at a minimum) to denote both the thing that is executing in some runtime environment and the thing that is its archetype. This distinction is sometimes (not always) expressed as component *instances* versus component *types*. This is what CCL does. In prose, we use the term "component" in some contexts to mean component type and in others to mean component instance. In CCL, however, the keyword **component** is used to introduce the specification of a component type.

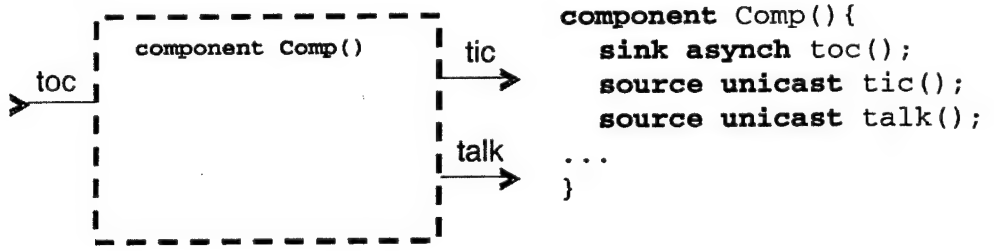


Figure 2: Structural Aspects of a Component

A component in CCL has a structural part and a behavioral part. Figure 2 shows the structural part of a component specification: its pin specification. From this specification, we know that instances of `Comp` will accept an interaction request on its asynchronous sink pin `toc` and react by requesting interaction on its source pins `tic` and `talk`.

For simplicity we have bypassed several features of CCL:

- There are no initialization parameters on the component type (or on any other instantiations in this example), as indicated by the empty `()` in `Comp()`.

- Only asynchronous interaction modes are used (as **asynch** and **unicast**).
- The pins do not exchange data, as denoted by the empty `()` in `toc()`, `tic()`, and `talk()`.

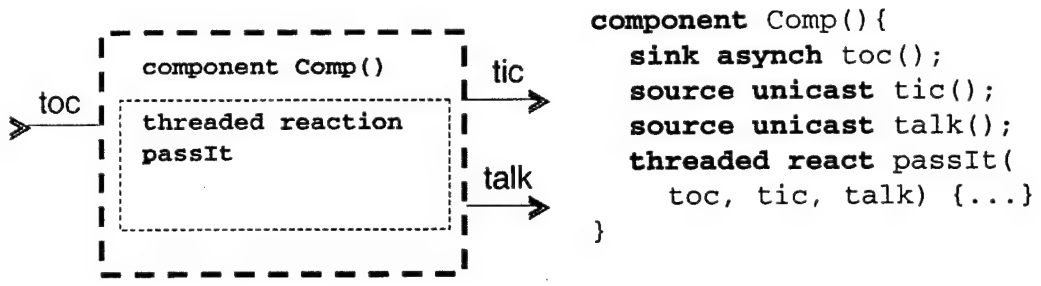


Figure 3: Behavioral Aspects of a Component

Figure 3 shows the preamble to the behavioral part of a component: its reaction(s), introduced with the CCL keyword **react**. A reaction specifies the externally visible behavior of a component, where "externally visible" means visible on the component's pins. More specifically, the behavior at each pin is specified in terms of two distinct kinds of events: start events, which begin interactions on a pin, and end events, which terminate interactions on a pin. Each event is denoted by a pin name and a start/end prefix operator, ^/$, respectively. For example, `^toc` represents the start of an interaction on the `toc` pin. (See Figure 4 for more examples.)

For a component to be well formed, each sink pin must be associated with *exactly* one reaction. This ensures that the reactive behavior of a component with respect to each sink pin is unambiguous. Another well-formedness rule is that each source pin is associated with *at least* one reaction. In this way, several reactions may request interactions with the same external resource. These rules are statically enforced by our CCL processor. The pin names in the parenthetical list (`toc`, `tic`, `talk`) of the `passIt` reaction appear to be parameters of the reaction but are in fact declarations of the pins used by the reaction.

Reactions may be either threaded or unthreaded (the default). A threaded reaction is introduced by prepending the CCL keyword **threaded** to **react**, as shown in Figure 3. Here, thread denotes the concept of a unit of concurrency rather than its implementation as, say, a Java thread. There are several distinct well-formedness rules governing threaded and unthreaded reactions. For example, **reenter** sink pins may be associated only with unthreaded reactions, and each reentrant sink pin must be the only sink pin of its reaction.

The details of reaction behavior are specified in a variant of UML statecharts (based on the UML 1.5 statechart semantics [OMG 02]). To statecharts we add an executable action language, of which only a portion—generating events—is illustrated. The CCL specification of

component Comp is completed in Figure 4. A summary of the syntax and semantics of statecharts is presented in the following paragraphs.

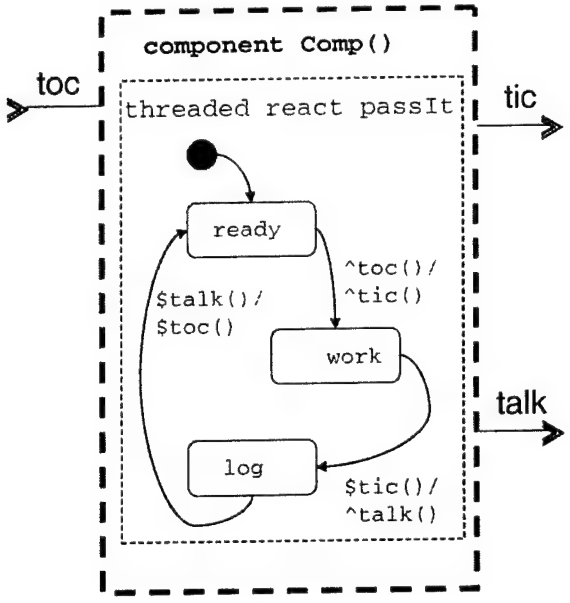


Figure 4: Statechart Specification of Reactions

```
component Comp(){
  sink asynch toc();
  source unicast tic();
  source unicast talk();
  threaded react passIt(
        toc, tic, talk) {
    start->ready{}
    ready->work{
      trigger ^toc();
      action ^tic();
    }
    work->log{
      trigger $tic();
      action ^talk();
    }
    log->ready{
      trigger $talk();
      action $toc();
    }
  }
}
```

A *transition* may

- optionally observe an event (CCL keyword **trigger**)
- optionally define a Boolean guard that is tested to determine whether a transition is eligible to fire (CCL keyword **when**)
- optionally execute one or more actions if the transition fires (CCL keyword **action**)

A *state* may specify actions that are executed

- on entry to the state (CCL keyword **enter**)
- when an outbound transition from the state fires (CCL keyword **exit**)

In the example, `passIt` waits in the `ready` state until it observes the start interaction event `^toc`. In response, `passIt` generates the `^tic` event to start an interaction with any component(s) connected to source pin `tic` and transitions to the `work` state. `passIt` waits in the `work` state until the interaction on `tic` completes, denoted by observing the `$tic` event, at which time it generates the `^talk` event to start an interaction on the source pin `talk` and transitions to the `log` state. `passIt` then waits in the `log` state until the interaction on `talk` completes, denoted by observing the `$talk` event. In response, `passIt` generates the `$toc`

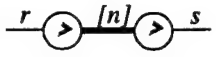
event to signal the termination of this interaction on `toc`, and returns to the `ready` state, where it waits for the next `^toc` event.

Many well-formedness rules are defined on reactions. One such rule is that transitions in a reaction may only observe ^*sink* events, where sink is the name of a sink pin, or *$source* events, where source is the name of a source pin. The dual to this rule is that only *$sink* events and ^*source* events may be generated as actions. This rule is easy to understand, and thereby unlocks the meaning of its dual: a reaction cannot force the component with which it is interacting to terminate that interaction; therefore a *$source* cannot be generated, only observed.

CCL Capsule: Pin Interaction Modes

Each pin obeys one of CCL's predefined interaction modes. There are two major modes: asynchronous and synchronous. Informally, they classify message-based and procedure-based modes of interaction. Modes are further specialized by pins.

Sink pins specialize synchronous mode into those that enforce mutual exclusion on reactions, denoted in CCL as **`sink mutex`**, *and those that permit concurrent behavior on reactions, in CCL* **`sink reenter`**. *Source pins specialize asynchronous mode into* **`unicast`** *and* **`multicast`**, *meaning about what you would expect. This graphic shows the iconographic conventions we use to depict most of these, for source pins r and sink pins s:*

	`source (unicast` *or* `multicast)` *and* `sink asynch`
	`source synch` *and* `sink mutex`

Our experience suggests that asynchrony on the source side may be specialized in many different ways; for example, for different queueing policies. Experience also suggests that the interplay of interaction modes and concurrency can be confusing at first exposure. For example, reentrant sink pins cannot be associated with threaded reactions because a thread of control can process only one request at a time, which implies that callers may have to wait until previous calls have been completed.

2.2 Specify the Environment

The runtime environment shares with components the "type versus instance" distinction. As with components, the term "environment" is used for both, but the CCL keyword **`environment`** is used to introduce an environment type specification.

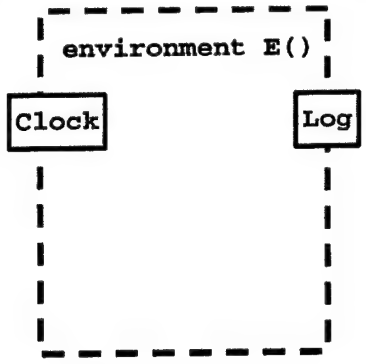
The two roles an environment plays in a component technology are reflected in CCL:

1. An environment provides services.
2. An environment provides interaction mechanisms.

Although the example illustrates only the first of these roles, both are discussed here. It is also worth noting that CCL specifications of environments, which in some ways are messier than those of components and assemblies, are intended to be supplied by the developers of PECTs supporting specific runtime environments, and hence to be reusable for multiple assemblies or systems.

2.2.1 Specify the Environment Services

Figure 5 specifies that environment E provides two services, Clock and Log. Services are nothing more than environment-provided components. Syntactically, their specifications differ only by the keyword **service**. A distinguishing syntactic characterization is provided because environment components may have different well-formedness rules than application components. As a concrete example of such a difference, services may execute behavior independent of the arrival of a stimulus, whereas application components must be purely reactive.[2]




```
environment E (){
  service Clock() {...}
  service Log() {...}
}
```

Figure 5: Structural Aspects of an Environment

Because we use the same syntax for services as for components, we need not discuss the details of the service specification shown in Figure 6. As earlier, we assume empty initialization parameters on the service types.

2. The motivation is to localize non-determinism to the environment. This restriction may prove too strong to be imposed generally.

```
environment E() {
  service Clock() {
    source unicast tic();
    threaded react ticking(tic){
      start->sleeping{}
      sleeping->pulsing{
        trigger after(10);
        action ^tic();
      }
      pulsing->sleeping{
        trigger $tic();
      }
    }
  }
  service Log() {
    sink asynch listen();
    threaded react
        logging(listen) {
      start->run{}
      run->run{
        trigger ^listen();
        action $listen();
      }
    }
  }
}
```

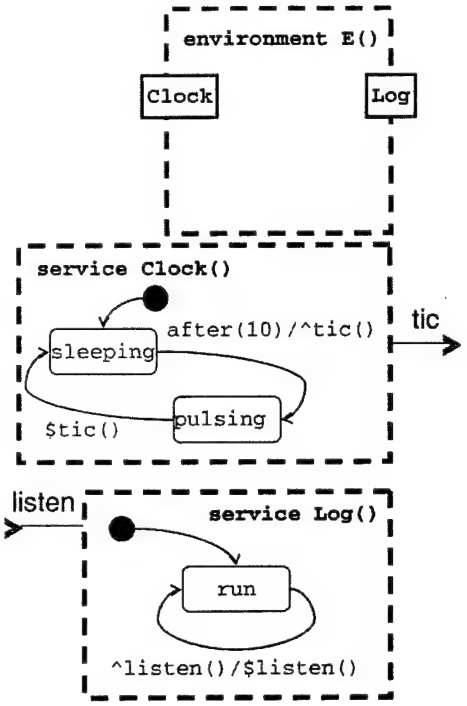


Figure 6: Environment Services

2.2.2 Specify Environment-Specific Interaction Mechanisms

Components interact with each other in different ways depending on the mechanism, or connector, used to enable their interactions. A synchronous mechanism (such as a function call) is very different from an asynchronous mechanism (such as sending a message); they have different consequences with respect to whether the initiator of the interaction will block until a response is received or even whether such a response is permitted. Consequently, when components are composed, the behavior of the composition does not depend solely on the behavior of the components; the behavior of the interaction mechanism is also relevant.

This means that the behavior of interaction mechanisms, like the behavior of components, must be explicitly modeled. As with reactions, we model the behavior of interaction mechanisms using statecharts. This allows us to define the meaning of a composition of components in terms of the statecharts for the reactions and those for the interaction mechanisms.

Modeling an interaction mechanism is not as simple as modeling a reaction, though. A simple approach with one statechart for each connection between a source pin and a sink pin is not always appropriate. When multiple source pins are connected to the same sink pin, there is often some coordination among these interactions, such as a single message queue that holds each request until it can be processed; such coordinations must be modeled also. Dividing the behavior of an interaction mechanism into two pieces—a **`source handler`** and a **`sink handler`**—allows us to separate interaction behavior into parts that are coordinated in different ways. Doing so simplifies composition.[3]

Each handler is defined with a statechart that describes how the interaction mechanism mediates communication from the point of view of one of the participants. Source handlers typically describe the conditions that allow the source's reaction to proceed; for example, whether the source reaction is blocked until the interaction is complete or only until the interaction request has been queued with a sink handler. Sink handlers typically describe how multiple requests for an interaction on the same sink pin are coordinated; for example, whether they are queued, and if so, what the queuing policy is.

Handlers differ from other language concepts in that they do not follow the same type/instance pattern. While handler types are defined explicitly (as sketched in the CCL fragment below), instances are not declared explicitly. Instead, tools algorithmically determine which handlers to use for a particular composition. The choice of handler types is based on the types of pins involved in an interaction (as indicated by the parenthetical list of pin types supported by each handler) and on whether handlers for different pins should be combined. For example, multiple asynchronous sink pins of a component that all use the same message queue should all use the same sink handler, which models that queue's effect on interactions.

```
Environment E() {
...
  sink handler (mutex) {...}
  sink handler (mutex, asynch) {...}
  source handler (unicast) {...}
...
}
```

Each handler statechart (which would appear where the ellipses are in the above CCL fragment) is actually an incomplete prototype that is algorithmically parameterized and modified when instantiated in the context of a particular composition (e.g., replacing generic `^sink` and `$source` events with specific events corresponding to the relevant pins, and adding additional transitions to accommodate multiple pins interacting with the same handler). We regard the algorithm by which the CCL processor selects the appropriate handlers, and how it per-

3. The way that CCL handles connectors—separating their models into handler models—differs from the way that many architecture description languages (ADLs) treat connectors. The same meaning—the mediation of interaction and protocols that participants must follow—is retained, but the separation gives us flexibility in composing behavioral models.

forms statechart composition, as a matter for a CCL *back end*.[4] For further details about statechart composition see "Preserving Real Concurrancy" [Ivers 03].

The means for capturing these points of variability in the definition of handlers are not yet fully defined, so complete handler models cannot be presented at this time.

2.3 Specify the Inner Assembly

Assemblies exhibit the type/instance distinction, just as components, environments, and services do. As in the previous cases, the keyword **`assembly`** introduces an assembly type. Also as before, the term "assembly" is sometimes used to denote both type and instance. However, the distinction between type and instance is a bit more complicated in the case of assemblies.

Assemblies are specified relative to some environment, which supplies the mechanisms that make component interaction within an assembly possible. That is, component interaction always uses environment-supplied interaction mechanisms, or connectors. Environments also provide services that may be useful in assemblies.

CCL Capsule: Actions

Actions in CCL are expressed for the most part in a very restricted subset of ANSI C. The choice of C syntax was made with the thought of using CCL in conjunction with industrial-strength software development environments—and for this purpose C/C++ is eminently suitable.

CCL currently supports **`int`** *and* **`boolean`**. *Simple type naming is allowed following the C* **`typedef`** *syntax, with restrictions. The built-in CCL* **`array`** *type abstracts the details of C pointers, which are not supported by CCL.*

Variable declarations, assignments, and expressions are, as in C, minus pointers. Event generation is shown with a syntax that looks almost like C procedure calls, with the difference of a start or end unary operator before each event name. The following operators are supported, with their C-defined precedence (except for operators not found in C, of course).

CCL Operator Precedence

`:`	*name scope (highest)*
`()`	*grouping*
`[ ]`	*array indexing*
`+, -, !, ^, $`	*unary plus, minus, negation, start, end*
`*, /, %`	*multiply, divide, modulus*
`+, -`	*add, subtract*
`<, <=, >, >=`	*less than, less than or equal, greater than, greater than or equal*
`==, !=`	*equal, not equal*
`&&, ||`	*logical and, logical or*
`=`	*assignment*

The `Inner` assembly specification begun in Figure 7 shows the relationship between assemblies and environments. The assembly (type) specification is parameterized by exactly one environment (type). Since we do not yet have an environment instance, we do not know how

4. The terms "back end" and "front end" refer to architectural roles played by different components of a traditionally structured compilation system. A front end typically performs lexical and syntactic (so-called "static semantic") analysis, while a back end might generate code for a target platform.

many Clock or Log service instances will be available. However, the assembly specification can introduce assumptions about how many runtime service instances (and of what service types) will be available with the CCL keyword **assume**. In this example, Inner assumes that an environment-provided instance of Clock is available, which is denoted as innerClock. An analogous assumption is made about the Log service.

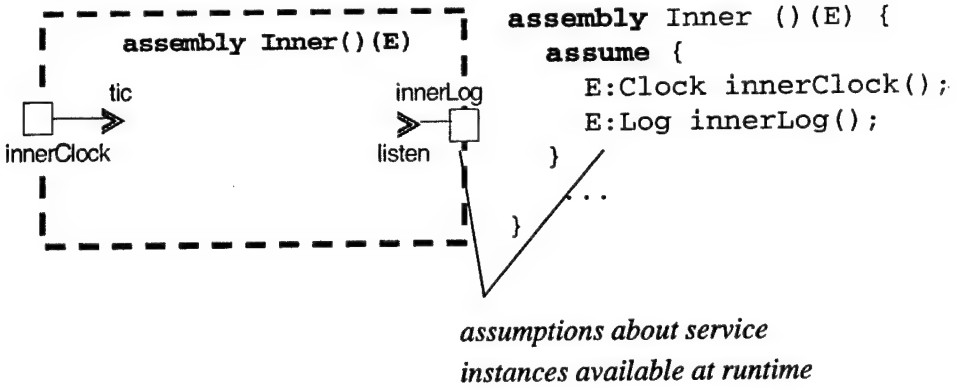
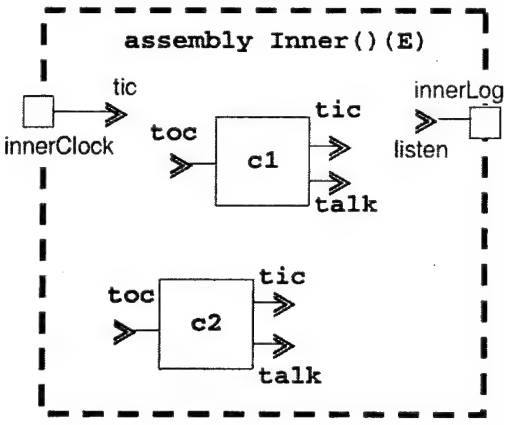


Figure 7: Assembly Instantiation and Service Assumptions

Instantiation syntax is used to declare innerClock and innerLog. In CCL, as in C, *instances* are *declared* using a "*type instance*" declarator syntax. The **assume** clause contains what looks like two instantiations, which, from the language user's perspective, are as good as instantiations. Semantically, however, innerClock and innerLog are pseudo instances only—they are merely assumptions that must be fulfilled later, when the assembly Inner is instantiated.

The service assumptions innerClock and innerLog are expressed in terms of service types defined in E. This is allowed because the prologue "**assembly** Inner () (E)" makes environment E visible in the scope of Inner. There must also be a set of components visible in the scope of Inner; otherwise, this is going to be a boring assembly. In CCL all components and environments are specified in the global namespace,[5] allowing us to add two instances of component Comp—c1 and c2—to Inner in Figure 8.

5. In fact, the environment E is visible to Inner without the parameterization of Inner by E. The parametrization is used to check that only services of the specified environment type can appear as assumptions in the assembly. CCL does not currently support a hierarchical namespace such that components and environments can be grouped together in useful ways, for example as collections of related components. Such a feature will be needed eventually.



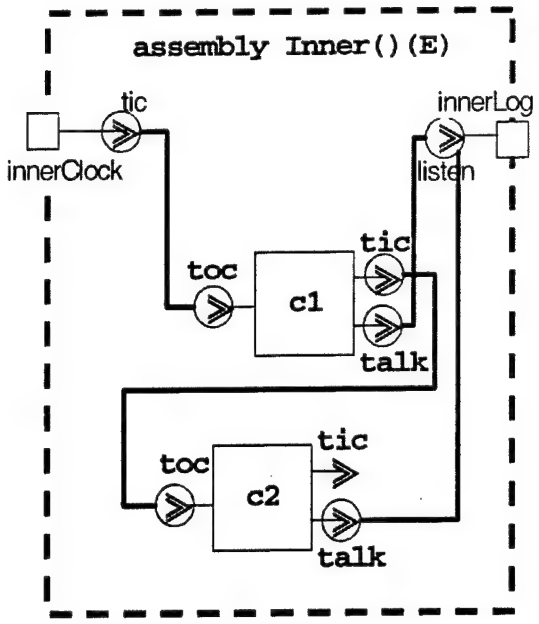

```
assembly Inner ()(E) {
  assume {
    E:Clock innerClock();
    E:Log innerLog();
  }
  Comp c1(), c2();//instances
  ...
}
```

Figure 8: Component Instantiation Within an Assembly

Components must be connected if they are to interact. The CCL ~> operator is used to connect a source pin (the left operand) to one or more sink pins (the right operands). Various conformance rules must be satisfied by connections, such as

- The source and sink pins must have compatible interaction types (e.g., **unicast source** and **asynch sink** are compatible, while **unicast source** and **mutex sink** are not).
- Each **consume** data parameter on the source pin must correspond positionally to a **produce** parameter on each sink pin, and the types of these parameters must be compatible (see the **produce** and **consume** capsule for more information on parameter modes).
- Each **produce** data parameter on the source pin must correspond positionally to a **consume** parameter on the sink pin, and the types of these parameters must be compatible.

Interactions must satisfy other such topological conditions as well, for example those imposed by a reasoning framework. Specifying well-formedness rules for a reasoning framework may also require additional (reasoning framework specific) properties to be supplied about assemblies. CCL provides an annotation mechanism for specifying such properties. For example, enforcing a priority ceiling emulation on a set of interacting components would require property annotations of thread priority. While reasoning framework specific well-formedness rules are not discussed further in this example, Figure 9 illustrates the use of the annotation mechanism to assign priorities to the components' reactions. Figure 9 also continues the illustration by specifying interactions among service and component instances.

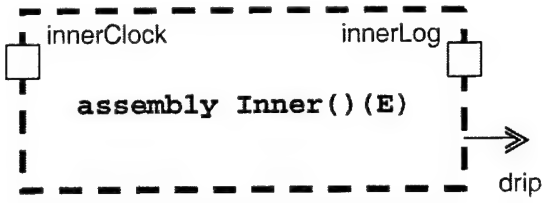



```
assembly Inner ()(E) {
  assume {
    E:Clock innerClock();
    E:Log innerLog();
  }
  //instances
  Comp c1(), c2();
  // properties
  annotate c1 {
    "passIt:priority", const
    int priority = 15}
  annotate c2 {
    "passIt:priority", const
    int priority = 10}
  // interactions
  innerClock:tic ~> c1:toc;
  c1:tic ~> c2:toc;
  c1:talk ~> innerLog:listen;
 c2:talk ~> innerLog:listen;
 ...
}
```

Figure 9: Component Interaction Within an Assembly

The last detail that must be specified about assemblies is the visibility of their interactive behavior. By default, assemblies have no visible interactive behavior. An assembly *A* that exposes no interactive behavior must execute independently from all other components and assemblies not contained in *A*. If an assembly instance *a* of type *A* is to interact with component (or assembly) instances not contained in *a*, we must expose, using the CCL keyword **expose**, the necessary pins (source or sink) of the components contained in *a*. These exposed pins are specified as a comma-separated list of scoped pin names; for example, the name `c2:tic` specifies the pin named `tic` of component instance `c2`.

The effect of `Inner` exposing the pin `c2:tic` is to define an alias of `c2:tic`, called `tic`, in the scope of `Inner` (i.e., `Inner:tic`). Renaming using the CCL keyword **as** followed by a name alias is sometimes necessary to preserve the name uniqueness of pins in assembly scope. For example, **expose** `{c1.tic, c2.tic}` would result in two pins named `tic` in the scope of `Inner`, which would be illegal. At least one of the exposed pins must be renamed; for example, as in **expose**`{c1.tic, c2.tic` **as** `drip}`.



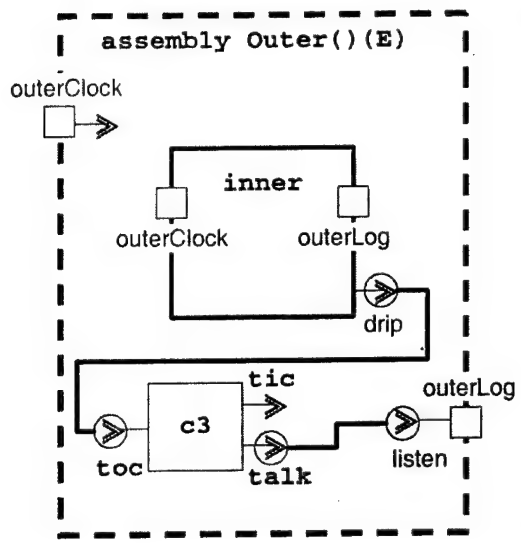

```
assembly Inner ()(E) {
  assume {
  E:Clock innerClock();
  E:Log innerLog();
  }
  //instances
  Comp c1(), c2();
  // interactions
  innerClock:tic ~> c1:toc;
  c1:tic ~> c2:toc;
  c1:talk ~> innerLog:listen;
  c2:talk ~> innerLog:listen;
  expose {c2:tic as drip}
}
```

Figure 10: Abstracting Interaction Details

Figure 10 completes the specification of `Inner` by exposing interactions on `c2:tic`, renamed to the external world as `drip`. The remaining behavior is hidden, resulting in the graphical representation found in Figure 10.

2.4 Specify the Outer Assembly

The specification of the `Outer` assembly uses the assembly specification syntax presented in Section 2.3, but introduces a new syntax to deal with assembly instantiation. (We have already seen the instantiation of components, without complication, in Figure 8.) The complete specification for the `Outer` assembly is provided in Figure 11. The part of CCL that deals with assembly instantiation is highlighted.




```
assembly Outer ()(E) {
  assume {
    E:Clock outerClock();
    E:Log outerLog();
  }
  // instances
  Comp c3();
  Inner inner(){
    Inner:innerClock = outerCloc
    Inner:innerLog = outerLog;
  };
  // interactions
  inner:drip ~> c3:toc;
  c3:talk ~> outerLog:listen;
  expose {}
}
```

Figure 11: Elaboration of Assembly Instances

There are two points to note in the instantiation of `Inner` as `inner`.

1. In Figure 7, `Inner` made assumptions about its environment. It is the responsibility of the instantiating context to satisfy these assumptions. That responsibility is discharged in the elaboration clause { . . . } that follows assembly instantiation. Within the braces—{ }—of this elaboration, the syntax LHS = RHS is interpreted as "the service assumption on the left- hand side (LHS) is satisfied by the service instance on the right-hand side (RHS)."[6] Note that in this example, `Inner` assumptions are satisfied by `Outer` assumptions, which is perfectly legal.

2. Although its meaning will not become clear until `Outer` is itself instantiated, `Inner` is instantiated (as what was referred to in Volume III) as a partial assembly. Concretely, this means that the instances of components contained by `inner` will execute in the instance of the runtime environment of the instantiating assembly (in this case, `Outer`). In this example, we have bypassed the complicating situations where `inner` executes in a different runtime instance `e` of environment `E`, or where `inner` executes in a runtime instance `e'` of some other environment `E'`, where $E \neq E'$.

6. The use of "=" is misleading here, since what is occurring is not quite assignment. An alternative syntax would be *LHS satisfies RHS*, where, in this case, RHS would be the assumption of the instantiated assembly, and LHS would be a service instance provided by the instantiating (assembly) context.

2.5 Instantiate the Application myApp

The instantiation of partial assembly `inner` shows how assemblies can be used to model hierarchical composition. An alternative form of instantiation is used to define a complete runtime view of an assembly.

That view specifies a set of component instances, executing in one or more environment instances.[7] In CCL this specification involves two steps:

1. Instantiate an environment `E` as `e` to serve as the runtime environment for one or more assembly instances.
2. Instantiate an assembly `A` as `a`, in `e` to serve as the application; `e` must satisfy the assumptions imposed by `A`.

***CCL Capsule:* `produce` *and* `consume` *versus* `in` *and* `out`**

The meaning of `in` *and* `out` *parameters on procedures (methods) seems obvious:* `in` *parameters are supplied by the caller while* `out` *parameters are supplied by the callee. But CCL uses* `produce` *and* `consume` *in place of the conventional* `in` *and* `out` *for pin parameter modes:*

```
sink mutex addOne (
   consume int i, produce int i_plus_one);
```

To understand this decision, consider what the meaning of `in/out` *pin parameters for sink and source pins would be if used with the typical meaning. For a sink pin, an* `in` *parameter would be supplied by the caller (another component), and therefore would be data coming into the pin's component. For a source pin, however, an* `in` *parameter would be supplied by the pin's component and would therefore be data going out of the component. In practice, this meaning gets a bit confusing.*

Instead, we use `produce` *and* `consume` *with the perspective always relative to a pin's own component. Consequently, a* `produce` *parameter on a* `sink` *pin is a parameter produced by that pin's component, just as a* `produce` *parameter on a* `source` *pin is a parameter produced by that pin's component. Both represent data flow out of the component, but the first flow is on the conclusion of an interaction, while the second is on the initiation of an interaction.*

2.5.1 Instantiate Environment E as e

Recall that services are defined in environments; or, for emphasis, environment types provide a scope for service types. In CCL, the instantiation of an environment and its services is joined in one syntactic statement, as in the following CCL fragment (building on the earlier specification of environment E):

```
E appEnv() // appEnv is an instance of E
  {        // elaboration
    E:Clock appClock();
    E:Log appLog();     // appClock, appLog in appEnv's scope
  };       // end elaboration and instantiation
```

7. Currently, component topology is fixed at instantiation time and is thereafter immutable. This is a reasonable assumption for many, but not all systems. Also, the association of component instance with environment instance is immutable: in particular, there is no "mobility" of component instances to different environment instances. These and other important context assumptions of CCL can be made explicit only with a more comprehensive and formal treatment of CCL.

As with assembly instantiation, environment instantiation involves an optional elaboration clause. Whereas the assumptions of an assembly instance about its runtime environment are recorded when it is instantiated, the service instances provided by an environment are recorded in its instantiation. In the above fragment, two service instances—`appClock` and `appLog`—are created and permanently associated with environment instance `appEnv`. Note that `appClock` and `appLog` are genuine instantiations, unlike those that appear in **assume** clauses.

2.5.2 Instantiate Assembly Outer as myApp

To bring the example to a close, component instances and their enabled interactions must be associated, once and finally, with their (containing) environment instances. Once again, this is achieved in CCL using the instantiation mechanism. This CCL specification fragment does the job:

```
Outer myApp() (appEnv)
  {  // elaboration
    Outer:outerClock = appEnv:appClock;
    Outer:outerLog = appEnv:appLog;
  }; // end elaboration and instantiation
```

The above instantiation of myApp differs in two fundamental ways from the instantiation of `inner` as a partial assembly in Figure 11.

1. `myApp` is permanently associated with its runtime environment, `appEnv`. More fully, all component instances within the scope of `myApp`, including the component instances found in the transitive closure of all partial assemblies contained by `myApp`, are associated permanently with `appEnv`.
2. Service instances of `appEnv` are used to satisfy the assumptions of `Outer`. Syntactically, this association of assumptions and provisions (or satisfiers) is identical to that used in the elaboration clause of partial assemblies. Semantically, this actually satisfies these assumptions rather than passing them along.

At this point we have completed the example, resulting in the iconography of Figure 1, whose corresponding complete CCL specification is provided in Appendix A.

3 Related Work

CCL builds on previous work in the related areas of module interconnection languages, architecture description languages (ADLs), coordination languages, and composition languages. A number of languages developed in these areas inspired our work and remain of interest. While developing CCL has provided us with an opportunity to address some shortcomings of such languages and explore new concepts for capturing the information needed to support predictable assembly, we can still benefit from reflecting on other work. In particular, now that the majority of the concepts behind CCL are well understood, we can reexamine and consider the significance of CCL's differences from these languages.

Some of the more prominent languages that we continue to consider are

- The Avionics Architecture Description Language (AADL): AADL incorporates many features from an earlier domain-specific ADL, MetaH [Honeywell 00], that are well suited to support some of the prediction technologies with which we are currently working [Feiler 03].
- Acme: Acme is an architectural interchange language that can also be used as an ADL in its own right due to its inclusion of necessary structural concepts and its flexible annotation mechanisms, which allow arbitrary extensions [Garlan 97].
- Darwin: Darwin is an ADL in which behavior is formally specified to support automated analysis [Magee 93].
- Koala: Koala combines ideas from the Darwin ADL with a component model for consumer electronics software. It includes a number of features addressing scalability and usability concerns motivated by industrial use that reflect needs we must also address [van Ommering 02].
- UML 2.0: As mentioned earlier, UML 2.0 will introduce new features that better capture architectural information and increase the similarities between UML and CCL. Given UML's widespread adoption, an encouraging transition path for PACC concepts may be to create a UML profile that captures the same information as CCL.

4 Current State and Next Steps

CCL has been a significant aid in developing our understanding of PECT. For example, we have developed a much better understanding of the difference between deploying and instantiating components; in fact, instantiation is the real background subject of this report. To date, the development of CCL has been driven as much (or more) by the inquiry into what belongs in any language that plays the role of CCL as it has been by considerations of syntactic elegance. As a consequence, it is fair to say that CCL has not achieved much in the way of such elegance. The authors fully expect the syntax of CCL to evolve, perhaps substantially, as end-user (and possibly automation) considerations become more apparent with experience.

One area of CCL semantics that has yet to stabilize is the treatment of connectors, as discussed in Section 2.2.2. Until this area of CCL becomes stable, we will have no way of specifying or analyzing interactions among components. There are also numerous useful but minor language features that remain to be implemented (such as arrays).

The CCL semantics are nowhere, outside of the current implementation of a CCL front end, explicitly specified. An external and, where feasible, formal specification of these semantics is necessary for many reasons. We are examining two approaches that are not mutually exclusive. One approach is to use a traditional technique for specifying language semantics—perhaps structured operation semantics. The other approach is to define a UML 2.0 "profile" for CCL or extend the UML metamodel to encompass CCL. The first approach provides better prospects for near-term development efforts; for example, the development of interpretations to model checkers and simulation environments. The second approach offers better prospects for integrating CCL with commercial UML environments; it also offers the potential for replacing the current implementation of CCL with one written in a more recent language-specification technology; for example, the Generic Modeling Environment (GME) [Ledeczi 01].

A functioning CCL front end is not sufficient: it must also export to its users—PECT developers in this case—the means for implementing interpretations. Minimally, this consists of a well-documented programming interface. The CCL front end currently exports an interface for an abstract syntax tree of the CCL language, with nodes bearing various semantic annotations previously computed and deposited by front-end semantic analysis. This interface is still under development and is exported as ANSI-C rather than the more programming-friendly ANSI-C++ or something language neutral. In the near term, a pragmatic "only when specifically needed" approach will be taken to improving the programming infrastructure. This, of course, will have implications on our ability to provide this implementation to third parties.

We have asserted elsewhere [Hissam 02] that the CCL processor is just one of various forms of automation used in developing and using a PECT. Design issues are raised by a language-centric (in this case, CCL-centric) development environment. There is, for instance, the matter of interfaces and interchange standards for different elements of the environment. And, use of a CCL-based environment will entail very practical considerations for managing components, component and assembly specifications, intermediate products (specifications in development), analysis results (predictions), and so forth. As before, we adopt a pragmatic "only when specifically needed" approach to this kind of infrastructure development.

Appendix A Complete Example

The following source was processed successfully by a CCL processor. Keywords are not presented in boldface.

```
// CCL specification of example 1
environment E() {
  service Clock() {
     source unicast tic();
     threaded react ticking(tic) {
        start->sleeping{}
        sleeping->pulsing{trigger after(10); action ^tic();}
        pulsing->sleeping{trigger $tic();}
      }
   }
   service Log() {
     sink asynch listen();
     threaded react logging(listen) {
        start->run{}
        run->run{trigger ^listen(); action $listen();}
     }
   }
}

component Comp() {
  sink asynch toc();
  source unicast tic();
  source unicast talk();

  threaded react passIt(toc, tic, talk) {
        start->ready{}
        ready->work{trigger ^toc(); action ^tic();}
        work->log{trigger $tic(); action ^talk();}
        log->ready{trigger $talk(); action $toc();}
  }
}
assembly Inner ()(E) {
   assume {
      E:Clock innerClock();
      E:Log innerLog();
```

```
    }
    Comp c1(), c2();  // instances

    // properties
    annotate c1 {"passIt:priority", const int priority = 15}
    annotate c2 {"passIt:priority", const int priority = 10}

   innerClock:tic ~> c1:toc;
    c1:tic ~> c2:toc;
    c1:talk ~> innerLog:listen;
    c2:talk ~> innerLog:listen;

expose {c2:tic as drip}
}

assembly Outer ()(E) {
    assume {
        E:Clock outerClock();
        E:Log outerLog();
    }
    Inner inner(){
        Inner:innerClock = outerClock;
        Inner:innerLog = outerLog;
    };
    Comp c3();

    inner:drip ~> c3:toc;
    c3:talk ~> outerLog:listen;

    expose {}
}
E appEnv() { E:Clock appClock(); E:Log appLog(); };

Outer myApp()(appEnv) {
   Outer:outerClock = appEnv:appClock;
   Outer:outerLog = appEnv:appLog;
};
```

Glossary[8]

annotation	a property P associated with a referent R, meaning that "R has property P," denoted as *R.P*
assembly	a set of components and their enabled interactions
component	an implementation in final form, modulo bound labels, that provides an interface for third-party composition and is a unit of independent deployment
connector	a mechanism provided by the runtime environment that enforces an interaction protocol or discipline on the components that are participants in an interaction
compose	to enable component interaction through connectors
composition	a set of interactions among components enabled through connectors. See also *assembly.*
contain	to restrict the visibility of interactions on pins. All interactions among components are restricted to the scope of the most immediately enclosing ("containing") assemblies and partial assemblies.
interaction	a composition of two or more reactions, from distinct components, using a runtime-environment-provided connector
interpretation	a mapping from assemblies specified in CCL to specifications in the language of a reasoning framework
partial assembly	a (recursively defined) abstraction that aggregates a set of components and their enabled interactions and exposes selected component pins. Logically, a partial assembly is a component implemented in terms of other components. See also *assembly.*

8. Adapted from *Volume III: A Technology for Predictable Assembly from Certifiable Components* [Wallnau 03].

property — an *n*-tuple <*name, value, ...* >, where *name* and *value* refer to the name of some property and the value it takes, respectively. See also *annotation*.

reaction — specification of the behavior of a unit of concurrency within a component (e.g., a thread) and the behavioral dependencies between the sink pins and source pins of a component

reasoning framework — a combination of a property theory, an automated reasoning procedure, and a validation procedure that is used to predict assembly properties

pin — binding labels in the construction and composition language (CCL). See also *source pin*, *sink pin*, *connector.*

runtime environment — environment that provides runtime services that may be used by components in an assembly, provides an implementation for one or more connectors, and enforces assembly constraints

sink pin — a pin that accepts interactions with the environment of a component (i.e., from other components or the runtime environment). See also *pin*, *source pin*.

source pin — a pin that initiates interactions with the environment of a component (i.e., to other components or the runtime environment). See also *pin*, *sink pin*.

References

URLs valid as of the publication date of this document.

[Feiler 03] Feiler, P.; Lewis, B.; & Vestal, S. "The SAE Avionics Architecture Description Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering." *Real-Time and Embedded Technology and Applications Symposium (RTAS) 2003 Workshop on Model-Driven Embedded Systems*. Washington, DC, May 27-30, 2003. <http://www.cse.wustl.edu/~cdgill/RTAS03/published /SAEAADLRTASv1.pdf> (2003).

[Garlan 97] Garlan, D.; Monroe, R.; & Wile, D. "Acme: An Architecture Description Interchange Language," 169-183. *Proceedings of CASCON '97*. Toronto, Ontario, November 10-13, 1997. Toronto, Canada: IBM Press, 1997. <http://www.cs.ubc.ca/local/reading /proceedings/cascon97/cascon97/htm/english/abs/garlan.htm>.

[Hissam 02] Hissam, S. & Ivers, J. *PECT Infrastructure: A Rough Sketch* (CMU/SEI-2002-TN-033, ADA413548). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. <http://www.sei.cmu.edu/publications/documents /02.reports/02tn033.html>.

[Honeywell 00] Honeywell Laboratories. *MetaH User's Manual*. Minneapolis, MN: Honeywell, Inc., 2000. <http://www.htc.honeywell.com /metah/uguide.pdf>.

[Ivers 02] Ivers, J.; Sinha, N.; & Wallnau, K. *A Basis for Composition Language CL* (CMU/SEI-2002-TN-026, ADA407797). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. <http://www.sei.cmu.edu/publications/documents /02.reports/02tn026.html>.

[Ivers 03] Ivers, J. & Wallnau, K. "Preserving Real Concurrency." *Correctness of Model-Based Software Composition (CMC) Proceedings*. Karlsruhe, Germany, July 22, 2003. <http://www.ubka.uni-karlsruhe.de/vvv/ira/2003/13/13.pdf> (2003).

[Ledeczi 01] Ledeczi, A.; Maroti, M.; Bakay, A.; Karsai, G; Garrett, J.; Thomason, C.; Nordstrom, G; Sprinkle, J.; & Volgyesi, P. "The Generic Modeling Environment." *IEEE International Workshop on Intelligent Signal Processing (WISP'2001).* Budapest, Hungary, May 24-25, 2001. <http://www.isis.vanderbilt.edu/Projects/gme/GME2000Overview.pdf>.

[Magee 93] Magee, J.; Dulay, N.; & Kramer, J. "Structured Parallel and Distributed Programs." *Software Engineering Journal 8,* 2 (March 1993): 73-82.

[OMG 02] Object Management Group. *OMG Unified Modeling Language Specification Version 1.5.* OMG document formal/03-03-01. <http://www.omg.org/technology/documents/formal/uml.htm> (September 2002).

[van Ommering 02] van Ommering, R. Part 5, Ch. 12, "The Koala Component Model," 223-236. *Building Reliable Component-Based Software Systems.* Crnkovic, Ivica & Larsson, Magnus, eds. Boston, MA: Artech House, 2002.

[Wallnau 03] Wallnau, K. *Volume III: A Technology for Predictable Assembly from Certifiable Components* (CMU/SEI-2003-TR-009). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. <http://www.sei.cmu.edu/publications/documents/03.reports/03tr009.html>.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (leave blank)	2. REPORT DATE June 2003	3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Snapshot of CCL: A Language for Predictable Assembly		5. FUNDING NUMBERS F19628-00-C-0003
6. AUTHOR(S) Kurt C. Wallnau, James Ivers		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2003-TN-025
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116		10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES		
12.a DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS		12.b DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

Construction and composition language (CCL) plays several roles in our approach to achieving automated predictable assembly. CCL is used to produce specifications that contain structural, behavioral, and analysis-specific information about component technologies, as well as components and assemblies in such technologies. These specifications are translated to one or more reasoning frameworks that analyze and predict the runtime properties of assemblies. CCL processors can also be used to automate many of the constructive activities of component-based development through various forms of program generation. Using a common specification for prediction and construction improves confidence that analysis models match implementations. This report presents a snapshot of CCL by examining a small example CCL specification.

14. SUBJECT TERMS composition language, predictable assembly, prediction-enabled component technology, PECT, component technology				15. NUMBER OF PAGES 40 16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED		20. LIMITATION OF ABSTRACT UL